Towards Flexible Task & Team Maintenance*

Ari Yakir and Gal A. Kaminka and Nirom Cohenov-Slapak
The MAVERICK Group
Computer Science Department
Bar Ilan University, Israel
{yakira,galk}cs.biu.ac.il

Abstract

There is significant interest in modeling teamwork in synthetic agents. In recent years, it has become widely accepted that it is possible to separate teamwork from taskwork, providing support for domain-independent teamwork at an architectural level, using teamwork models. However, existing teamwork models (both in theory and practice) focus almost exclusive on *achievement goals*, and ignore *maintenance goals*, where the value of a proposition is to be maintained over time. Such maintenance goals exist both in taskwork (i.e., agents take actions to maintain a condition while a task is executing), as well as in teamwork (i.e., agents take actions to maintain the team). This paper presents DIESEL, an implemented teamwork and taskwork architecture, built on top of Soar, that addresses maintenance goals in situated agent teams. We provide details of DIESEL's structure, and initial experiments demonstrating it in operation in a dynamic rich domain.

Introduction

There is significant interest in modeling teamwork in synthetic agents, for training (Rickel & Johnson 1999) simulation (Tambe 1997), robotics (Parker 1998; Kaminka & Frenkel 2005), and entertainment (Tambe *et al.* 1999; D.Vu *et al.* 2003). In recent years, it has become widely accepted that it is possible to separate teamwork from taskwork, providing support for domain-independent teamwork at an architectural level, using teamwork models.

*This research was supported in part by ISF grant #1211/04
Copyright © 2006, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

Existing teamwork models are informed by studies of teamwork in humans, and have focused on a subset of teamwork components, such as task allocation and synchronized selection and termination of joint goals. These models enjoyed considerable success in such environments, due to the significant reuse opportunities they offer, and the robustness of the resulting system.

However, existing models only account for a subset of phenomena associated with human teamwork. Specifically, existing teamwork models (both in theory and practice) focus almost exclusive on *achievement goals*, where the value of a proposition is to be changed from its current settings to another. Agents form a team and agree on a task to be executed (goal to be reached, i.e., proposition to hold in some future state), and then dissolve the team once the task is completed. Sequences of tasks are carried out by constant dissolving and re-formation of the team in question, per task (Tambe & Zhang 1998).

Human and synthetic teams, however, also tackle *maintenance goals*, where the value of a proposition is to be maintained over time. Such maintenance goals exist both in taskwork (i.e., agents take actions to maintain a condition while a task is executing), as well as in teamwork (i.e., agents take actions to maintain the team). Examples of maintenance goals in teamwork include robust service maintenance (Kumar, Cohen, & Levesque 2000) and continual task allocation (Parker 1998). Examples of maintenance goals in taskwork includes continual information sharing and monitoring for robotic formations (Kaminka & Frenkel 2005) and in entertainment (D.Vu *et al.*

2003).

This paper addresses maintenance goals in situated agent teams. We developed DIESEL, an implemented teamwork and taskwork architecture, built on top of Soar (Newell 1990). The result allows agents to collaboratively maintain task-execution conditions, and teamwork-structure conditions, throughout the execution of a task. To demonstrate DIESEL's contribution, we carried out experiments in the GameBots domain (Kaminka *et al.* 2002), evaluating the use of collaboratively-maintained maintenance conditions in contrast to existing approaches.

Motivation and Related Work

We use a simple team task to motivate our work, especially in context of previous research in teamwork. In this task, a team of synthetic agents follows a leader that moves around, at a fixed distance. This is a simplified version of familiar robotic formation-maintenance tasks, e.g., as reported in (Kaminka & Frenkel 2005).

Most teamwork architectures to date have focused on achievement goals. Of those, most closely related to our work is the BITE architecture for multi-robot teamwork (Kaminka & Frenkel 2005). In BITE, agents automatically communicate with their teammates during allocation and synchronization, which take place when selecting and terminating behaviors for execution. However, BITE does not allow the specification of any team-reorganization, nor does it allow to specify any maintenance actions. BITE behaviors are seen as one compact block, thus, there is no way to specify any logic between the beginning and end of behaviors. So while the leader will start and stop with its followers, this will happen when the agents trigger end-conditions or preconditions for selected controllers. There is no way to actually have the agents take joint actions (through BITE) to maintain distances while behaviors are executing, e.g., by preventing failures. In other words, all maintenance is in fact carried out individually, instead of collaboratively.

CAST (Yen *et al.* 2001) addressed the issue of proactive information exchange among teammates, using an algorithm called DIARG, based on petri net structures. CAST shows the importance of team communication regarding information that might assist task achievement for individual members in a proactive manner, and aim to reduce communication. This approach, based on the theory of Joint Intentions, does not include maintenance of goals. In particular, CAST's communications focus on informing other teammates of discovered facts that may trigger preconditions. The use of communications (or other actions) to maintain currently existing tasks is not addressed.

ALLIANCE (Parker 1998) is a behavior-based control architecture focused on robustness, in which robots dynamically allocate and re-allocate themselves to tasks, based on their failures and those of their teammates. ALLIANCE offers continual dynamic task allocation facilities, which allocate and re-allocate tasks to agents while they are collaborating. It uses fixed teams, in the sense that addition and removal of robots from the team is handled by human intervention and it assumes that robots can monitor their own actions, and those of others. Our work differs in that we focus on maintenance not only of assignment of agents to tasks, but also of the joint execution itself.

STEAM (Tambe 1997) was a key step in teamwork architectures, implemented in Soar (Newell 1990). STEAM focused for the most part on achievement goals, similarly to BITE. However, a first step towards extending STEAM in terms of maintenance goals was introduced in (Tambe & Zhang 1998). In this work, four categories of teams are introduced. PTPM, a persistent team consisting of persistent members, PTNM, a persistent team consisting of non-persistent members, NTPM, a temporary form of a team consisting of persistent members and NTNM, a temporary form of a team consisting of non-persistent members. This work was the first to discuss reorganization (team hierarchy maintenance) in a team.

In (Tambe & Zhang 1998) persistence is referred as the degree of commitment of agents to the team, while executing a task. To enable such persistent teams, they use a decision-theoretic technique. In particular, their agents reason about expected team utilities of future team states. To accommodate real-time constraints, this reasoning is done in an any-time fashion. This approach left many open questions. First, their work only deals with maintenance of the team but not with maintenance in task execution. Second, due to the way the STEAM architecture works, reasoning about team states is not done in parallel to task execution, thus main-

taining the team structure must be done separately to mission execution.

DIESEL, described in this paper, deals with PTPM teams, i.e., persistence of team structure. We refer to this as teamwork maintenance. However, in contrast to (Tambe & Zhang 1998), DIESEL also addresses maintenance in tasks (which STEAM did not address). Moreover, it proposes a single mechanisms for both, and offers flexibility to the designer in deciding on protocols and behaviors to be used proactively and reactively.

Kumar and Cohen (Kumar, Cohen, & Levesque 2000; Kumar & Cohen 2000) extended the theory of Joint Intentions in order to include maintenance as a part of it. They define maintenance goals as follows: *if the agent does not believe p, it will adopt the goal that p be eventually true*. The maintenance goal is persistent (PMtG) if this fact remains true for the agent at least until the agent either believes that it is impossible to maintain p or that the maintenance goal is irrelevant.

$(PMtG\ x\ p\ q) \equiv [\neg(BEL \times p) \subset (GOAL \times \Diamond p)] \wedge$
$(UNTIL[(BEL \times \Box\neg p) \vee (BEL \times \neg q)]$
$[\neg(BEL \times p) \subset (GOAL \times \Diamond p)])$

Further elaborations for persistence can be seen in their Theorem 4.2.1: *If an agent having a persistent maintenance goal for p comes to believe not p, it will adopt a persistent achievement goal (PGOAL) for p in addition to the persistent maintenance goal*.

Persistent achievement goal is thus defined as:
$(PGOAL \times p) \equiv (BEL \times \neg p) \wedge (GOAL \times \Diamond p) \wedge$
$(UNTIL[(BEL \times p) \vee (BEL \times \Box\neg p)](GOAL x \Diamond p))$

While we build on the theoretical developments of (Kumar, Cohen, & Levesque 2000; Kumar & Cohen 2000), our work differs significantly, in several ways. First, unlike previous work, we extend maintenance of team structure to hierarchical teams, including team-subteam relations. We also address goal maintenance in hierarchical task decomposition. Second, our implementation allows for arbitrary, context-dependent protocols for collaborative goal maintenance. Finally, while Kumar and Cohen's work has been applied to teams of web services, our focus is on modeling synthetic humans in virtual environments.

Maintenance In Teamwork

We propose a new architecture that allows the automation of maintenance both of the team structure and of the behavioral structure. Our architecture extends structures common to STEAM (Tambe 1997), MONAD (D.Vu *et al.* 2003), and BITE (Kaminka & Frenkel 2005).

We use Soar for the implementation of our architecture. Soar is a general cognitive architecture for developing systems that exhibit intelligent behavior (Newell 1990). Soar uses globally-accessible working memory, and production rules that test and modify this memory. Efficient algorithms maintain the working memory in face of changes to specific propositions. Soar operates in several phases, one of which is a decision phase in which all relevant knowledge is brought to bear to make a selection of an operator, that will then carry out deliberate mental (and sometimes physical) actions. A key novelty in Soar is that it automatically recognizes situations in which this decision-phases is stumped, either because no operator is available for selection (*state no-change impasse*), or because conflicting alternatives are proposed (*operator tie impasse*). When impasses are detected, a subgoal is automatically created to resolve it. Results of this decision process can be chunked for future reference, through Soar's integrated learning capabilities. Over the years, the impasse-mechanism was shown to be very general, in that general problem-solving strategies could be brought to bear for resolving impasses.

Our architecture is composed of four structures:

1. A behavior graph, that defines the recipe structure by which agents achieve their goals. This is described below in detail.
2. A team hierarchy tree, that defines the organizational structure and chain of command.
3. A set of domain-independent reusable task-maintenance behaviors, referenced by the recipe.
4. A set of domain-independent reusable team-maintenance protocols, referenced by the team hierarchy.

These structures are described below in detail.

Guided by existing work, we chose hierarchical behaviors (in Soar terms, *operators*) as the basis for our representation and for the underlying controllers of the team members. Each behavior has

preconditions which enable its selection (the agent can select between enabled behaviors), termination conditions (which determine when its execution must be stopped, if previously selected) and application code containing the actual code for execution while the behavior is still running.

DIESEL additionally allows adding task-maintenance to each behavior. Task-maintenance conditions can be a conjunction or disjunction of predicates (referred to as events), needed to be maintained or denied by the team throughout the execution of a behavior. If a maintenance condition is broken, the maintenance mechanism will propose a specific maintenance behavior (in fact, a maintenance recipe) best suited to deal with such an occurrence. This will be done while executing the original behavior.

Behavior graph. Behaviors are arranged in a behavior graph, a graphical representation of a recipe for execution. A behavior graph is a connected, directed graph, where vertices denote behaviors. Vertical edges signify *decomposition* (i.e., from a behavior to sub-behaviors needed to execute it); horizontal edges signify *temporal ordering*, from a behavior to those that should ideally immediately follow it. The following is a simple recipe, as represented in Soar.

```
(<state> ^recipe <r>)
(<r> ^name long-corridor
     ^root <r1>)
(<r1> ^name root
      ^child <r2>
      ^child <r3>)
(<r2> ^name explore-decision
      ^first true
      ^child <r4>
      ^child <r5>
      ^next <r3>)
(<r4> ^name elaborate-target
      ^first true)
(<r5> ^name elaborate-no-target
      ^first true)
(<r3> ^name explore-movement
      ^child <r7>)
(<r7> ^name movement-to-target
      ^first true)
```

We used the conventions in (Kaminka & Frenkel 2005; Tambe 1997) to visualize the recipe above,

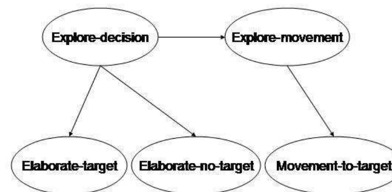

Figure 1: **An example recipe.**

as shown in Figure 1. This recipe, defines a behavioral tree hierarchy with a root node, and two child nodes called **explore-decision** and **explore-movement**. Explore-decision differs from explore-movement by being a first child type node. First child nodes can be proposed for execution right after their parent node is, and only if he is still active. Since explore-movement is not a first child type node it cannot be proposed simultaneously with its brother node (explore-decision). Consequently, it can be chosen only as a next type node. Next type nodes can be proposed only after a previous node (behavior) succeeded in achieving proper endconditions. In the given example, explore-movement will be proposed by the recipe mechanism only after explore-decision endconditions were achieved. Several next nodes can be proposed to deal either with success or failure of each previous node. This can be done by extracting data regrading the cause resulting in termination from a given endcondition.

Team hierarchy. To keep track of which behaviors are coordinated with which team-members, each behavior is tagged with a team name, which refers to a unique node in a team hierarchy representing team-subteam relations. For instance, a tag on behavior <*cn*> will look like this:

```
(<cn> ^team agent-group)
```

will cause the behavior to be synchronized by DIESEL within a team or sub-team called **agent-group**. Synchronization here is meant in the sense that selection, maintenance, and termination of those behaviors is automatically coordinated with all teammates.

The following is an example of such a simplified team hierarchy.

```
(<r0> ^name agent-group
```

```
        ^members <m0>
        ^coc <coc0>)
(<m0> ^name bot1)
(<coc0> ^list <l1>)
(<l1> ^name bot1)
```

The hierarchy above defines a team called agent-group with only one agent called *bot1*. *bot1* is defined as the sole coordinator of the team. He is the only team member listed in the chain of coordinators list named ˆcoc. The ˆcoc list is composed by task or subtask teammates which are the current teammates participating in the current task or subtask. The ˆcoc list usually contains a long list of optional coordinators which represent alternatives teammates for coordinating the task.

Maintenance behaviors . We have described above the introduction of maintenance conditions to each behavior. Such conditions can typically be maintained in one or two ways: By taking proactive actions to maintain the condition true; and by taking reactive actions when the condition becomes false.

Though the use of maintenance conditions in integrated architectures is rare, the key novelty in DIESEL is the ability to tie specific *team behaviors* to these conditions, and to specify the way they are to be used: proactively or reactively. The behaviors will be triggered automatically by DIESEL, to be executed by the associated team or subteam.

For example, suppose a task behavior that moves the agents around has a maintenance condition on it to maintain visual tracking of the leader. Because the behavior is a team behavior, it will be executed by the leader and the follower jointly. As a result, both leader and follower are mutually responsible for maintaining the condition. The maintenance behavior is itself a team-behavior, to be executed jointly by the leader and follower even as they are executing the task behavior (i.e., move around). An example of such a maintenance behavior may have the leader continually communicate its current position, and the follower orienting itself towards this position.

In our recipe in DIESEL, this is done by adding the following three lines to the *explore-decision* and *explore-movement* behaviors.

```
(<cn> ^maintenance <m2>)
(<m2> ^name see-leader)
(<m2> ^type negative-logic)
```

Tagging see-leader maintenance as negative-logic, means that maintenance actions for the see-leader event will be proposed on top of a specific behavior, as long as it is present.

Teamwork maintenance. Just as task-execution behaviors can have associated maintenance conditions, so can the team hierarchy be maintained by the use of team-maintenance conditions. As in the behavior hierarchy, these conditions are a set of conjunctions and disjunctions of predicates (referred to as events), needed to be maintained or denied throughout the execution of a task. Since maintenance operators act in order to maintain a possible team state, they are suited to allow team reconfiguration, all under the same teamwork mechanism. For example, if during the execution of a recipe sub-tree it is critical to maintain the number of teammates in the group fixed, such a team-maintenance condition could be easily defined, and the teamwork mechanism, can act in turn if such a condition fails, by joining a new team, recruiting new agents or even merging two teams. All whilst continuing the execution of the mission.

Evaluation

To evaluate the contribution offered by DIESEL, we build a small two-agent team in the GameBots domain (Kaminka *et al.* 2002), an adversarial game environment that enables qualitative comparison of different control techniques (e.g., (D.Vu *et al.* 2003)). In the first set of experiments, we focused on evaluating the effect of DIESEL's technique on the number of context-switches between behaviors, in comparison with architectures such as BITE (Kaminka & Frenkel 2005). We demonstrate show how team reconfiguration occurs using the same mechanism both on team hierarchy and behavior hierarchy.

In all experiments, we used the same recipe (Figure 1), with minor changes needed for each scenario discussed. The recipe consists of exploration and movement. During the exploration phase (behavior *explore-decision*), one of the two child behaviors can be proposed: *elaborate-no-target* in case there is no available target present, and *elaborate-target* in case there are one or more.

In the first case, the agent will tilt its pan-zoom camera, scan or rotate, and in the second case, a behavior will summarize target data, and propose all available options. *explore-decision*'s endcondition is that a target has been selected. Respectively, this is *explore-movement*'s precondition. In this case, a child behavior will be in charge of all movement actions taken by the agent in order to reallocate itself to a given target location.

Maintenance versus Achievement Goals

We will begin by first focusing on the general issue of supporting collaborative maintenance goals in a behavior-based architecture. The first set of experiments compares evaluates the use of individual maintenance goals and associated maintenance operators. A second set of experiments examines the difference between such individual goals and collaborative maintenance goals.

Individual maintenance goals. Our first experiment examines DIESEL's explicit support for maintenance goals, using the idea of task-maintenance behaviors that execute in parallel to the task-achievement behaviors. In this experiment, two agents are placed side by side on one end of a long corridor, closed off by a wall at one end. One agent is a leader, the other a follower. The leader runs until reaching the wall and then runs back. The follower's task is to run after the leader.

We are prohibiting any communications at this stage, since the task is purely individual for now. The follower will scan until it sees the leader, and run towards it. During each time tick, if the follower agent sees the leading agent, an internal event (*see-leader*) is fired and logged.

Figure 2 shows the event's occurrence during ten simulation runs. Each simulation's duration was about two minutes in real-time, 9000 decision-sense-act Soar cycles and fifty seconds in Unreal Tournament clock-time. In the figure, The X-axis shows the time in Unreal Tournament. The Y-axis separates the ten trials: Each dot shows the presence of the see-leader event in memory, at the give time, for the given trial. The figure shows that in all the experiments conducted without maintenance, after a short period of time, the follower lost the leader. This is due to the change in direction of the leading agent (back to the start location after

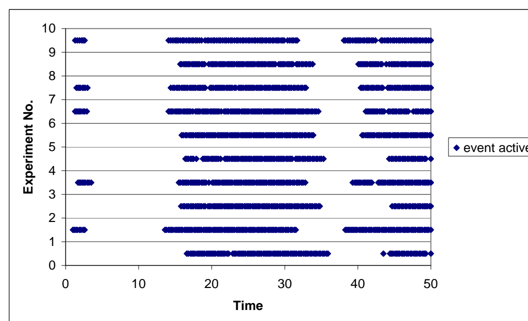

Figure 2: ***see-leader*** **event logged by the follower agent. No maintenance conditions.**

reaching the wall) which occurred during the follower's movement. In addition, sometimes when the follower agent locates the leader right away, it is only for a short period of time. This is due to the fact that the leader agent chose its target and began moving towards it, exiting from the follower's line of sight before the follower had a chance to react. This forced the follower agent to switch behavior, and re-locate its target.

Teamwork maintenance. Just as task-execution behaviors can have associated maintenance conditions, so can the team hierarchy be maintained by the use of team-maintenance conditions. As in the behavior hierarchy, these conditions are a set of conjunctions and disjunctions of predicates (referred to as events), needed to be maintained or denied throughout the execution of a task. Since maintenance operators act in order to maintain a possible team state, they are suited to allow team reconfiguration, all under the same teamwork mechanism. For example, if during the execution of a recipe sub-tree it is critical to maintain the number of teammates in the group fixed, such a team-maintenance condition could be easily defined, and the teamwork mechanism, can act in turn if such a condition fails, by joining a new team, recruiting new agents or even merging two teams. All whilst continuing the execution of the mission.

Figure 3 shows 10 additional trials, this time when an explicit maintenance condition was put in place. Here, we added an individual task-

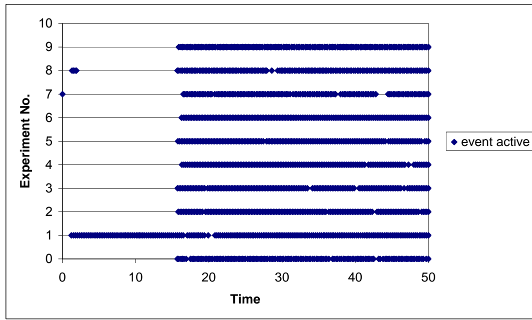

Figure 3: **Maintenance of the *see-leader* event by the follower agent.**

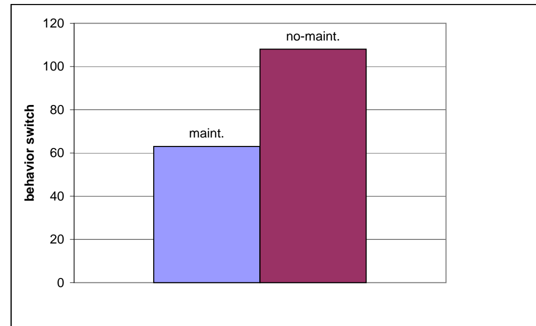

Figure 4: **The number of behavior switches with and without maintenance behaviors.**

maintenance condition to the recipe of the follower, instructing it to keep the leader in focus while moving. This is done by adding the following three lines to explore-decision and explore-movement behaviors.

```
(<cn> ^maintenance <m2>)
(<m2> ^name see-leader)
(<m2> ^type negative-logic)
```

Tagging see-leader maintenance as negative-logic, means that maintenance actions for the see-leader event will be proposed on top of a specific behavior, as long as it is present.

Figure 3 shows that now, the follower agent no longer loses track of the leader, since it actively pans to track the leader. This is an example of how task related goals can be set apart from maintenance related goals, adding new flexibility to behavior-based architecture and clarity to the code: It was achieved without changing the *explore-movement* or *move-to-target* behaviors, allowing to keep them simple and compact.

One desired outcome of using maintenance behaviors in parallel to task execution is that the number of behavior switches is significantly reduced. This allows for greater use of context in Soar and similar architectures, reducing thrashing. Figure 4 shows an the decrease in behavioral switches during the tests conducted. The figure shows the number of behavior switches in with and without maintenance behaviors.

Collaborative maintenance goals. These results show the importance of maintenance during behavior execution. However, one could point out that no teamwork is really being tested in these scenarios since no communication or coordination takes place, and argue that by the use of a teamwork architecture such differences could be solved.

To evaluate DIESEL's support for collaborative maintenance of goals, we chose a more complex environment for our agent team, one that would raise additional points of failure. Requiring the agents to explicitly collaborate on a given recipe is straight-forward in DIESEL: We simply add the following lines, on the behaviors we wished synchronized in the recipe:

```
(<cn> ^team agent-group)
```

This invokes DIESEL's team-operators, modeled after those in (Kaminka & Frenkel 2005; Tambe 1997), in order to carry out task allocation and synchronization of agents during the task execution. We chose a square-shaped corridor, in which the leader could run indefinitely. With every turn, the leader could potentially be blocked from the view of the follower, the agents had many opportunities to lose each other.

In previous work, the fact that the agents are explicitly collaborating in this task means that they will come to agreement as to the joint goal. Here, this is carried out by the leader announcing its next selected target point. Thus the leader move around, announcing its changing target locations as soon as it selects them. The follower agent can visually track the leader, and and can also run to the announced locations. As seen in Figure 5, the per-

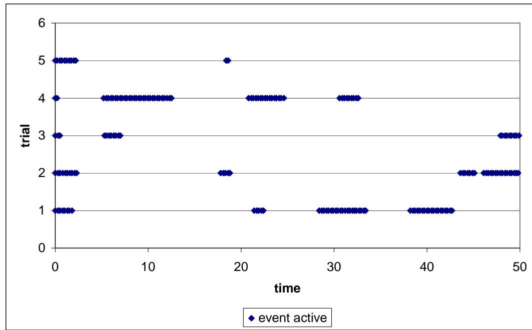

Figure 5: **see-leader event logged by the follower agent while using teamwork and communication**

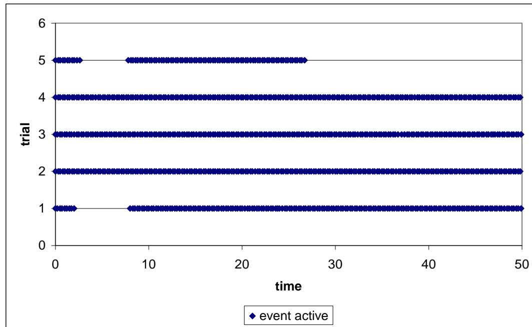

Figure 6: **maintenance on see-leader event while using teamwork and communication**

formance of the team using this technique is not impressive: The leader is rarely visually seen by the follower.

Figure 6 shows the performance of the follower agent using maintenance, in contrast to the results shown in Figure 5. Here, the team was jointly responsible for keeping the leader in sight during most of the mission. This was achieved by both agents taking active actions in order to maintain their formation. The two agents executed a protocol in which the follower proactively broadcast its *see-leader* event status, and also panned towards the leader. The leader agent monitored these broadcasts, and stopped to wait for the follower if the broadcasts stopped (i.e., it reactively maintained the condition). The figure shows significant improvement in *see-leader* maintenance durations.

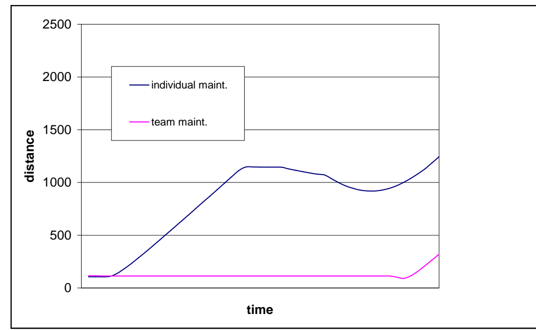

Figure 7: **Distance between leader and follower, in cases of individual and team goal maintenance.**

We stress the difference between individual and collaborative maintenance goals. In individual form, the leader would not be responsible for maintenance of the goal, and it would be up to the follower to carry out all actions necessary to maintain the distance. In collaborative maintenance, both leader and follower share the burden for maintaining the goals of the team. In DIESEL, this is achieved without changing the original recipe or behaviors.

To see this, we manually introduced a failure into the scenario above, where the follower was physically blocked from moving forward. While the follower agent conducts negative-maintenance, meaning it actively seeks to maintain the presence of see-leader events, the leading agent conducts a positive-maintenance, meaning it acts only when such an event drops. In this failure case, once the follower stopped tracking the leader, the leader's positive-maintenance is proposed (even while it was heading to its designated target), and commanded it to wait.

Figure 7 shows the results of such a case. The figure shows on the X-axis the passage of time (in Unreal Tournament seconds). The Y-axis shows the distance between the follower and leader. With individual maintenance, the distance between leader and follower continue to grow after the failure occurs. However, with team maintenance, distance between both agents is kept throughout the artificially-introduced failure.

Teamwork Maintenance

The previous section has focused on maintenance goals in the context of the task. One novelty in DIESEL is that it re-uses the same mechanism for maintaining the team hierarchy in face of catastrophic failures to individual agents. We call this teamwork-maintenance, to contrast with task-maintenance described in the previous section..

To demonstrate team-maintenance, we divided four agents into two groups, each consisting of a leader and a follower. We defined a single team-maintenance condition in each team, stating that each agent should have a coordinator at any given moment. In each team, the coordinator was initially set to be the leading agent. In team A, consisting of bot1 and bot2, it was bot1, and in team B, consisting of bot3 and bot4, it was bot3. This was part of the team-hierarchy for each agent. Both teams followed the same recipe previously described, with the two leaders independently leading their respective followers in constant movement along the corridor.

To show teamwork maintenance in action, we deliberately blocked any contact with bot3 and hid him during the first half of the experiment. As a result, bot4, changed its coordinator, and began following bot1, by joining team A. After running half of the experiment in such a manner, we removed the blocking on the original coordinator, bot3, thus allowing bot4 to fall back to its original team, team B.

Switching teams in this example is achieved by a team-maintenance behavior (operator, in Soar), which manipulates the chain of coordinators list present in bot4's team-hierarchy. The behavior works by checking whether at any given time a coordinator is unreachable. If so, then the behavior finds a new team in which there is a team coordinator and change the organizational membership of the agent to be a part of the other team. Since this is only a maintenance behavior, as opposed to a regular one, if the exception is resolved, the maintenance behavior is terminated, and regular order is restored.

In Figure 8 we can see an example run, in which we as can be seen,

The maintenance condition for this operator

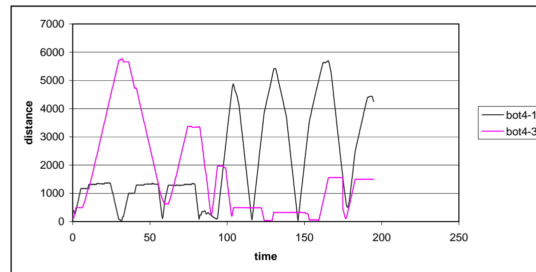

Figure 8: **Maintenance of team hierarchy: Distance between bot4 and bot3, bot1.**

Conclusions and Future Work

We presented DIESEL, an implemented teamwork architecture built on top of the Soar cognitive architecture (Newell 1990). Compared to previous work, DIESEL provides a novel single mechanism for collaborative maintenance of task goals as well as team structure. This allows the programmer to focus more clearly on achievement and maintenance aspects of the task, and to separate completely the issue of how to maintain the team-structure in face of catastrophic failures.

We provided results from initial trials using DIESEL for simple team tasks in the GameBots environment, and demonstrated that it results in reduced behavior switching and improved coordination between the agents. Future work includes extending the evaluation to more realistic, complex, tasks, and exploring a diverse set of maintenance protocols for taskwork and teamwork.

References

D.Vu, T.; Go, J.; Kaminka, G. A.; Veloso, M. M.; and Browning, B. 2003. MONAD: A flexible architecture for multi-agent control. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-03)*.

Kaminka, G. A., and Frenkel, I. 2005. Flexible teamwork in behavior-based robots. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*.

Kaminka, G. A.; Veloso, M. M.; Schaffer, S.; Sollitto, C.; Adobbati, R.; Marshall, A. N.; Scholer, A.; and Tejada, S. 2002. GameBots: A flexible

test bed for multiagent team research. *Communications of the ACM* 45(1):43–45.

Kumar, S., and Cohen, P. R. 2000. Towards a fault-tolerant multi-agent system architecture. In *Proceedings of the Fourth International Conference on Autonomous Agents (Agents-00)*, 459–466. Barcelona, Spain: ACM Press.

Kumar, S.; Cohen, P. R.; and Levesque, H. J. 2000. The adaptive agent architecture: Achieving fault-tolerance using persistent broker teams. In *Proceedings of the Fourth International Conference on Multiagent Systems (ICMAS-00)*, 159–166. Boston, MA: IEEE Computer Society.

Newell, A. 1990. *Unified Theories of Cognition*. Cambridge, Massachusetts: Harvard University Press.

Parker, L. E. 1998. ALLIANCE: An architecture for fault tolerant multirobot cooperation. *IEEE Transactions on Robotics and Automation* 14(2):220–240.

Rickel, J., and Johnson, W. L. 1999. Animated agents for procedural training in virtual reality: Perception, cognition, and motor control. *Applied Artificial Intelligence* 13:343–382.

Tambe, M., and Zhang, W. 1998. Towards flexible teamwork in peristent teams. In *Proceedings of the Third International Conference on Multiagent Systems (ICMAS-98)*.

Tambe, M.; Adibi, J.; Al-Onaizan, Y.; Erdem, A.; Kaminka, G. A.; Marsella, S. C.; and Muslea, I. 1999. Building agent teams using an explicit teamwork model and learning. *Artificial Intelligence* 111(1):215–239.

Tambe, M. 1997. Towards flexible teamwork. *Journal of Artificial Intelligence Research* 7:83–124.

Yen, J.; Yin, J.; Ioerger, T. R.; Miller, M. S.; Xu, D.; and Volz, R. A. 2001. CAST: Collaborative agents for simulating teamwork. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-01)*, 1135–1144.